Determining Equivalence
- **Goal:** eliminate redundant computations
  - Sparse conditional constant propagation:
    - Eliminates multiple computations
    - Eliminates unnecessary branches
  
  \[
  \begin{align*}
  x - 2y & \rightarrow x = 2y \\
  x - 1 & \rightarrow x = 4z \\
  k &= 1 + x \rightarrow k = 4z
  \end{align*}
  \]

- Can we eliminate equivalent expressions without constants?

Common Subexpression Elimination
- Recognizes textually identical (or commutative) redundant computations
- Replaces second computation by result of the first

\[
\begin{align*}
\alpha &= \beta + 2 \\
\gamma &= \delta + 3 \\
\epsilon &= \zeta + \eta
\end{align*}
\]

- How do we do this efficiently?

Value Numbering
- Each variable, expression, and constant: unique value number
- Same number \( \Rightarrow \) computes same value
- Based on information from within block
- Use hash functions to compute these

Computing Value Numbers
- Assign values to variables
  - \( a = 3 \Rightarrow \text{value}(a) = 3 \)
- Map expressions to values
  - \( a = b + 2 \Rightarrow \text{value}(a) = \text{hash}(+, \text{value}(b), 2) \)
- Use appropriate hash function
  - Plus: commutative
    - \( \text{hash}(+, \text{value}(b), 2) = \text{hash}(+, 2, \text{value}(b)) \)
  - Minus: not commutative
    - \( \text{hash}(-, \text{value}(b), 2) \neq \text{hash}(-, 2, \text{value}(b)) \)
Value Numbering Summary

- Forward symbolic execution of basic block
- Each new value assigned to temporary
  - Preserves value for later use even if original variable rewritten
    - \( a = x+y; a = x+z; b = x+y \)
    - \( t = a; a = a+z; b = t \)
- Maps
  - Var to Val
    - specifies symbolic value for each variable
  - Exp to Val
    - specifies value of each evaluated expression
  - Exp to Tmp
    - specifies tmp that holds value of each evaluated expression

Map Usage

- Var to Val
  - Used to compute symbolic value of \( y \) and \( z \) when processing statement of form \( x = y + z \)
- Exp to Tmp
  - Used to determine which temp to use if \( \text{value}(y) + \text{value}(z) \) previously computed when processing statement of form \( x = y + z \)
- Exp to Val
  - Used to update Var to Val when
    - processing statement of the form \( x = y + z \), and
    - \( \text{value}(y) + \text{value}(z) \) previously computed

Computing Value Numbers, Example

<table>
<thead>
<tr>
<th>Original Basic Block</th>
<th>New Basic Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = x+y )</td>
<td>( a = x+y )</td>
</tr>
<tr>
<td>( b = x+z )</td>
<td>( t1 = a )</td>
</tr>
<tr>
<td>( c = x+z )</td>
<td>( t2 = b )</td>
</tr>
<tr>
<td>( d = x+y )</td>
<td>( e = t2 )</td>
</tr>
<tr>
<td>( x \rightarrow v1 )</td>
<td>( v1+v2 \rightarrow v3 )</td>
</tr>
<tr>
<td>( y \rightarrow v2 )</td>
<td>( v3+v4 \rightarrow v5 )</td>
</tr>
<tr>
<td>( a \rightarrow v3 )</td>
<td>( v5+v6 \rightarrow v6 )</td>
</tr>
<tr>
<td>( b \rightarrow v4 )</td>
<td></td>
</tr>
<tr>
<td>( c \rightarrow v5 )</td>
<td></td>
</tr>
</tbody>
</table>

Interesting Properties

- Finds common subexpressions even if they use different variables in expressions
  - \( y = a+b; x = b; z = a+x \)
  - \( y = a+b; t = y; x = b; z = t \)
- Finds common subexpressions even if variable that originally held the value was overwritten
  - \( y = a+b; x = b; y = 1; z = a+x \)
  - \( y = a+b; t = y; x = b; y = 1; z = t \)

Problems

- Algorithm has a temporary for each new value
  - \( a = x+y; t1 = a \)
- Introduces
  - lots of temporaries
  - lots of copy statements to temporaries
- In many cases, temporaries and copy statements are unnecessary
  - Eliminate with copy propagation and dead code elimination

Global CSE

- Value numbering eliminates some subexpressions but not all
  - \( \text{Global CSE} \)
  - \( \text{value} \) value is not always equal to \( j \)’s or \( k \)’s value
Available Expressions

- Global CSE requires computation of available expressions for blocks b:
- Expressions on every path in cfg from entry to b
- No operand in expression redefined
- Then use appropriate temp variable for used available expressions

Available Expressions: Dataflow Equations

- For a block b:
  - AEin(b) = expressions available on entry to b
  - KILL(b) = expressions killed in b
  - EVAL(b) = expressions defined in b and not subsequently killed in b

Next Time

- Partial Redundancy Elimination
- Read ACDI:
  - Ch 13

Available Expressions, Example

- Build control-flow graph
- Solve dataflow problem
  - Initialize AEin(i) = universal set of expressions
  - AEin(b) = \cap_{i \in \text{Pred}(b)} AEout(i)
  - AEout(b) = EVAL(b) \cup (AEin(i) \setminus KILL(i))

Value Numbering Example

- Step 1: insert temps for conditionals

<table>
<thead>
<tr>
<th>Statement</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1; j = 2; k = 3;</td>
<td>1; 2; 3;</td>
</tr>
<tr>
<td>x = i; y = j; z = k;</td>
<td>1; 2; 3;</td>
</tr>
<tr>
<td>s1 = x + y; s2 = x * z;</td>
<td>3; 6;</td>
</tr>
<tr>
<td>s3 = s1 + s2; s4 = s1 * s2;</td>
<td>9; 18;</td>
</tr>
</tbody>
</table>

Value Numbering Example

- Step 2:
  - Add entry for each rhs
  - Remove entry when dependent variable changes

<table>
<thead>
<tr>
<th>Statement</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1; j = 2; k = 3;</td>
<td>1; 2; 3;</td>
</tr>
<tr>
<td>x = i; y = j; z = k;</td>
<td>1; 2; 3;</td>
</tr>
<tr>
<td>s1 = x + y; s2 = x * z;</td>
<td>3; 6;</td>
</tr>
<tr>
<td>s3 = s1 + s2; s4 = s1 * s2;</td>
<td>9; 18;</td>
</tr>
</tbody>
</table>